CMPSC 465 — Data Structures & Algorithms
Spring 2024 — Mehrdad Mahdavi and David Koslicki

Worksheet 4

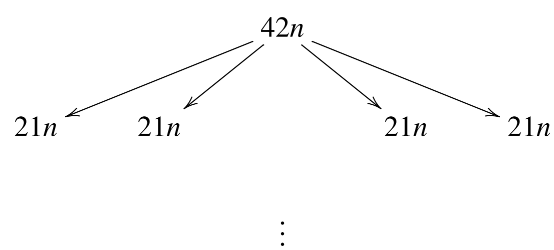
Wednesday, February 7, 2024

1. **Recurrence Relations.** Give the big-Θ bound for each of the following recurrence relations. You may assume the Master Theorem gives a big-Θ bound (the proof is similar to the big-O proof shown in class) and that $T(1) = O(1)$. Justify each of your answers.

   a) $T(n) = 4T(n/2) + 42n$
   b) $T(n) = T(n-1) + n^3$
   c) $T(n) = 7T(n/2) + n^2 + \log n$
   d) $T(n) = T(n/3) + 5$

   Solution:

   a) $T(n) = 4T(n/2) + 42n$
   Use the master theorem. Or:

   The first level sums to $42n$, the second sums to $84n$, etc. The last row dominates, and we have $\log n$ rows, so we have $42 \cdot 2^{\log n} \cdot n = \Theta(n^2)$.

   b) $T(n) = T(n-1) + n^3 = \sum_{i=1}^{n} i^3 + T(0) = \Theta(n^{3+1}) = \Theta(n^4)$

   c) Applying Master Theorem, we see that $a = 7$, $b = 2$, and $d = 2$. Since $\log_b a = \log_2 7 \approx 2.81 > 2$, we get $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7})$.

   d) Note that $5 = \Theta(1)$. We can apply the Master Theorem with $a = 1$, $b = 3$, $d = 0$. Therefore, $T(n) = \Theta(n^d \log n) = \Theta(\log n)$.

2. **Recurrence Relations.** What are the running times of each of these algorithms in big-O notation? Which is the fastest?

   a) Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.

   b) Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.

c) Algorithm C solves problems of size n by dividing them into eight subproblems of size $\frac{n}{4}$, recursively solving each subproblem, and then combining the solutions in $O(n^3)$ time.

Solution:

a) $T(n) = 5T(\frac{n}{2}) + O(n)$. This is a case of the Master theorem with $a = 5, b = 2, d = 1$. As $\log_b a > d$, the running time is $O(n^{\log_b a}) = O(n^{\log_2 5}) = O(n^{2.33})$.

b) $T(n) = 2T(n-1) + C$, for some constant C. $T(n)$ can then be expanded to $C\sum_{i=0}^{n-1} 2^i + 2^n T(0) = O(2^n)$.

c) $T(n) = 8T(\frac{n}{4}) + O(n^3)$. Thus we have $a = 8, b = 4, d = 3$. As $\log_b a < d$, we get $O(n^3)$

The sorted order by complexity will be $A < C < B$. Therefore, *Algorithm A will be the fastest.*

3. Merge. *A k-way merge operation.* Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

a) Here's one strategy: Using the `merge` procedure, merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of k and n?

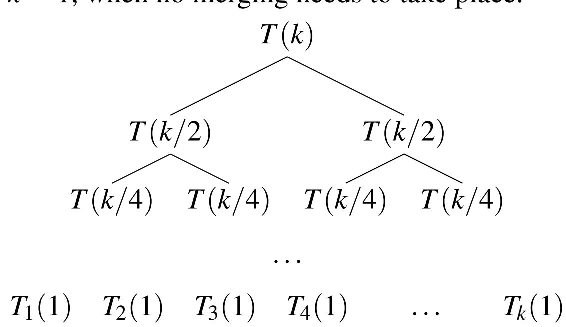
b) Give a more efficient solution to this problem, using divide-and-conquer.

Solution:

a) Each merge step requires $O(x+y)$ time for x elements in first array and y elements in the second array. Extending this, we can see that initially we start with $2n$ being the time required to merge the first two arrays. Then to merge with the third array it would take $3n$, and then $4n$ for the fourth and so on. Thus it can be represented as:

$$T(n) = 2n + 3n + 4n \ldots (k-1)n + kn = n(2 + 3 + 4 \ldots (k-1) + k) = n\left(\frac{k(k+1)}{2} - 1\right) = O(nk^2)$$

b) Recursively divide the arrays into two sets, each of $k/2$ arrays. Then merge the arrays within the two sets and finally merge the resulting two sorted arrays into the output array. The base case of the recursion is $k = 1$, when no merging needs to take place.



For each level, one can see that it will be $O(nk)$. Thus, the running time is given by $T(k) = 2T(k/2) + O(nk)$. Unrolling gives $T(k) = O(k) + \sum_{i=1}^{\log k} O(nk) = O(nk \log k)$.

Another possible approach to solve this problem without using divide-and-conquer is to merge all k arrays at the same time. To do this, you need an efficient way to decide which of the arrays has the next smallest element. This can be accomplished with a min-heap. Simply store the smallest elements (which are also the first elements) from each array in a min-heap, along with the array they came from and which index they occupied in that array. To fill the output array, delete the minimum element in the heap in $O(\log k)$ time, go to that element's original array, and insert the next value into the heap in $O(\log k)$ time. This process must be repeated n times, so the total running time is $O(nk \log k)$, the same as the divide-and-conquer approach.

4. **Array Rotations.** Consider a rotation operation that takes an array and moves its last element to the beginning. After n rotations, an array $[a_0, a_1, a_2 \ldots a_{m-1}]$ of size m where $0 < n \leq m$, will become:

$$[a_{m-n}, a_{m-n+1}, \ldots, a_{m-2}, a_{m-1}, a_0, a_1, \ldots a_{m-n-1}]$$

Notice how a_{m-1} is adjacent to a_0 in the middle of the new array. For example, two rotations on the array $[1,2,3,4,5]$ will yield $[4,5,1,2,3]$.

You are given a list of unique integers `nums`, which was previously sorted in ascending order, but has now been rotated an unknown number of times. Find the number of rotations in $O(\log n)$ time. *(Hint: consider Binary Search.)*

Solution:

Given an array like $arr = [10,1,2,3,4,5]$. We can use a modified version of binary search where in after we get the middle, we have to choose the side containing the pivot point. This side can be determined by the following:

1. When pivot is on the left half, then we know that the start of the array will be greater than middle i.e. $arr[start] > arr[mid-1]$
2. Else if pivot is on the right half then the middle will be of a greater value than the end $arr[mid+1] > arr[end]$

Pseudo code:

while $start < end$ **do**
  $mid \Leftarrow \lfloor (start+end)/2 \rfloor$
  if $arr[mid] > arr[mid+1]$ **then**
    return $mid+1$
  end if
  if $arr[mid-1] > arr[mid]$ **then**
    return mid
  end if
  if $arr[start] > arr[mid-1]$ **then**
    $end = mid-1$
  else
    $start = mid+1$
  end if

end while

5. **More Recurrence Relations.** Give the big-Θ bound for each of the following recurrence relations. You may assume that $T(1) = O(1)$. Justify each of your answers.

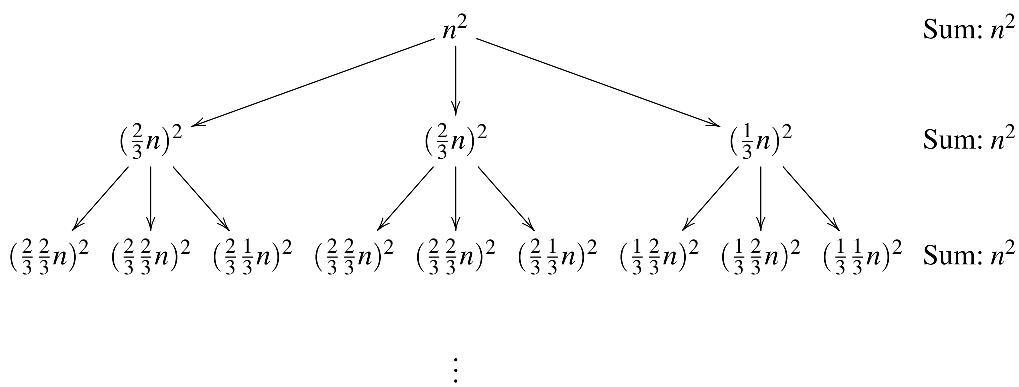
   a) $T(n) = 2T(2n/3) + T(n/3) + n^2$

   b) $T(n) = 3T(n/4) + n\log n$

   Solution:

   a) $T(n) = 2T(2n/3) + T(n/3) + n^2$

   If you visualize it as a tree you get:



   The height of the tree is $\Theta(\log n)$ because n is being divided in each step. Since there is n^2 work being done on each level, the final answer is $\Theta(n^2 \log n)$.

   b) $T(n) = 3T(n/4) + n\log n$

   On expanding the recurrence we get

$$T(n) = n\log n + 3T(n/4) = n\log n + 3(\frac{n}{4}\log\frac{n}{4} + 3T(\frac{n}{16})) = n\log n + 3\frac{n}{4}\log\frac{n}{4} + 9\frac{n}{16}\log\frac{n}{16} + \ldots$$

   Thus, we can see that we end up with $\sum_{i=0}^{\log_4 n}(\frac{3}{4})^i n\log\frac{n}{4^i}$. We can lower-bound this by $n\log n$ by taking the first term, and upper-bound it by $n\log n$ be replacing $\log(n/4^i)$ by $\log n$, so this is $\Theta(n\log n)$.